John von Neumann Institute for Computing

Pack Transposition: Enhancing Superword Level Parallelism Exploitation

C. Tenllado, L. Piñuel, M. Prieto, F. Catthoor

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata (Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 573-580, 2006.

http://www.fz-juelich.de/nic-series/volume33

Pack Transposition: Enhancing Superword Level Parallelism Exploitation*

C. Tenllado[a], L. Piñuel[a], M. Prieto[a], F. Catthoor[b]

[a]Dpto. de Arquitectura de Computadores y Automática
Universidad Complutense, 28040 Madrid, Spain
e-mail: {tenllado, lpinuel, mpmatias}@dacya.ucm.es

[b]IMEC
Kapeldreef 75, B-3001 Leuven, Belgium
e-mail: catthoor@imec.be

Current compilers do not allow for an efficient exploitation of the *Superword Level Parallelism* (SLP) available in many image processing applications. In this paper we present a modified version of the SLP compiler algorithm introduced by Larsen, that overcomes some of these problems. As motivating examples we have considered two kernels extracted from multimedia application, in which state-of-the-art compilers fail to exploit the available fine-grain data parallelism. Our methodology manages to extract the available SLP and achieves consistent speedups on both an Intel Pentium 4 and an Apple G4-based platform.

1. Introduction

Contemporary computer applications are multimedia-rich, involving significant amounts of audio, video and image processing. This trend has resulted in a variety of new ISA extensions, usually denoted as *Multimedia Extensions*, to practically all general-purpose microprocessors, including IBM-Motorola's AltiVec [3] for PowerPC, and Intel's MMX, SSE1, SSE2 and SSE3 [4,5] for Pentium. While different processors vary in the type and number of multimedia instructions offered, at the core of each is a set of short SIMD style operations. These instructions boost the performance of media applications operating concurrently on data that are packed in advance in a single wide (vector) register.

Unfortunately, these extensions do not provide yet transparent performance improvements. Despite the early success of automatic vectorization for traditional vector supercomputers, state-of-the-art vectorizing compilers for multimedia extensions have yet to demonstrate their effectiveness. In fact, applications developers usually turn to explicitly hand-tune their codes using in-line assembly, intrinsic functions or specialized library routines.

Classic approaches for automatic vectorization, such as the Allen-Kennedy algorithm [6], are based on the theory of data-dependence analysis, which was developed during the 70's and the 80's for array-based Fortran programs from the scientific computing domain. Dependence analysis is used to detect loop statements that could be executed in parallel without violating the semantics of the program (vector loops), and loop transformations, such as loop interchange or loop fission, were developed to increase such occurrences.

The similarity with vector processors has prompted the adaptation of classic approaches to compile for multimedia extensions. However, differences in both the target architecture (especially in the memory system) and the target domain make this adaptation difficult. General-purpose architectures

*This work has been supported by the Spanish goverment research contract TIC 2002-750 and the Hipeac European Network of Excellence

only support efficiently accesses to adjacent memory addresses, and only on vector length aligned boundaries. Although *Media Extensions* usually provide mechanisms to reorganize data elements in vector registers to deal with such situations (packing, unpacking and special shuffle instructions), they are not easy to use, and incur considerable penalties. On the other hand, traditional loop-based vectorization has been focused on statically analyzable codes, where little or no dynamic behavior is present, whereas multimedia codes are no longer static and make extensive use of pointers, making data dependence analysis more complex.

Our research is based on the *Superword Level Parallelism* compiler, an alternative approach introduced by Larsen et al. in [8]. It is focused on packing isomorphic instructions from the same basic block to vector instructions, and can be seen as a restricted form of *Instruction Level Parallelism* (ILP). We have extended the Larsen's compiler with additional transformations aimed at extracting the available SLP in a more efficient way.

The rest of the paper is organized as follows. In Section 2 we describe a motivating example to illustrate the problems under scope. Section 3 summarizes the original SLP compiler. Section 4 introduces our proposed enhancements, analyzing in Section 5 their efficiency on two different architectures, a Power-PC platform and an Intel Pentium4. Finally the paper ends with some conclusions.

2. Motivating example

A motivating example extracted from 2D image processing [2] is illustrated in Figures 1 and 2. The same algorithm is applied first to the image rows and then to the image columns. The array is scanned row by row in both cases due to some locality optimizations performed in early steps of the compilation process.

```
for i=0 to N-1
   for j=0 to N-3
      A[i,j+1] = α*A[i,j]+β*A[i,j+2];
```

Figure 1. A simple 1D algorithm applied on rows.

```
for i=0 to N-3
   for j=0 to N-1
      A[i+1,j] = α*A[i,j]+β*A[i+2,j];
```

Figure 2. A simple 1D algorithm applied on columns.

In the situation described by Figure 2, vector style parallelism is easily exploited by state-of-the-art approaches, since all the dependencies are carried out by the external loop [10,8]. However, they become inefficient when the dependencies are carried out by the inner loop (Figure 1). A classic vector compiler would apply loop interchanging [10], but this is not appropriate in our context since, it produces an access pattern with poor spatial locality and elements to be packed are not stored in adjacent memory addresses. The larsen's compiler would unroll the inner loop, which allows for a partial vectorization.

Is it possible to extract more SLP in cases similar to the one described in Figure 1? In this paper we try to answer this question. Basically, we propose some modifications to the SLP compiler algorithm that allow us to efficiently exploit the vector parallelism available in the outer loop.

3. Overview of the SLP compiler

Before describing the SLP compiler, it is convenient to remind some definitions introduced in [8]:

Definition 3.1 *A Pack is an n-tuple,* $\langle s_1, s_2, s_3, \cdots, s_n \rangle$*, where* $s_1, s_2, s_3, \cdots, s_n$ *are independent isomorphic statements in a basic block.*

Definition 3.2 *A PackSet is a set of Packs.*

Definition 3.3 *A Pair is a Pack of size two, where the first statement is considered the left statement, and the second statement is considered the right element.*

Definition 3.4 *The SuperWord Size (*sws*), is the maximum number of data elements that can be packed in a short vector register on the target platform.*

The SLP compiler extracts parallelism from the innermost basic block in a loop nest. It is based on a pre-processing, in which the innermost loops are unrolled by a factor equal to the sws. This unrolling constructs a basic block with several consecutive instances of the same statement. If vector parallelism could be extracted from the original loop, it can now be extracted from the basic block.

The core of the SLP compiler is applied later on a three-address representation of the code. It is subdivided into four phases: *Adjacent Memory Identication*, *PackSet Extension*, *Combination* and *Scheduling*, which are described bellow.

In the *Adjacent Memory Identication* stage, the basic block is scanned searching for *Pairs* with adjacent memory references, which are grouped together forming the initial *PackSet*. Adjacency is determined using both alignment information and array analysis. No *Pairs* are formed that cross alignment boundaries.

Statements can belong simultaneously to two *Pairs* as long as they occupy the left and right positions in the two *Pairs* respectively. This allows the *Combination* stage to easily merge groups into larger clusters. A simple example of the process is described in Figure 3.

More *Pairs* are added to the *PackSet* in the next stage. The compiler does it following the *use-def* and *def-use* chains of the *Pairs* that are currently in the *PackSet*. In this way the new members will consume superwords already formed or will provide the ones needed for an existing *Pair*. In all cases alignment consistency is checked.

Once all the possible candidates have been discovered, the *combination* stage is started. Here two *Pairs* are combined if the right statement of the first *Pair* is the same than the left statement of the other (Figure 3). The combined *Pairs* form a *Pack*. This process continues till no further combination is possible. The alignment consistency guaranties that the *Packs* formed will never cross alignment boundaries and that its size will not exceed the sws.

Finally the *PackSet* contains potential *Packs* of statements that can be executed in parallel using SIMD extensions. Nevertheless, it could happen that executing two groups of statements in parallel produces a dependency violation. A dependency cycle among *Packs* indicates an invalid set of groups, being necessary to remove at least one of the *Packs*.

After *Scheduling*, every *Pack* in the *PackSet* corresponds to a SIMD instruction, and possibly additional pack/unpack instructions. Refer to the original paper [8] for more details.

The weakest points of the SLP compiler are:

- It cannot efficiently extract SLP when dependencies are carried by the inner loop.
- The statement Packing is not steered. It could potentially be enhanced if the packing process is performed according to some information extracted from dependence analysis.
- It does not consider the chance of combining superwords to obtain new seeds for the *Packing* process. The number of *Packs* finally created could be larger taking this into account.

Some of these points are covered in our methodology for the kind of algorithms under scope.

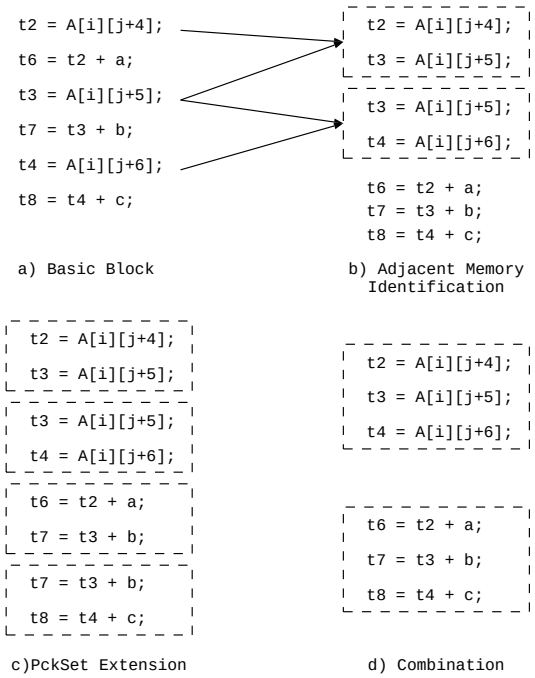

Figure 3. Simple example to illustrate the original SLP methodology.

4. Methodology description

Our methodology, which we have denoted as PT-SLP: *Pack Transposition* SLP, targets two level loop nests that operate on two dimensional arrays with the following constraints:

- The inner loop iterates over the lowest dimension of the arrays (assuming a row-major layout).
- All the dependences in the loop nest are carried by the inner loop. As dependences we consider flow, anti, output and input dependences.
- 2D structures represented by 1D arrays are accessed by affine functions

For this kind of algorithms, the loop unrolling performed by Larsen's compiler is not enough to uncover the vector parallelism. Alternatively, we propose to use first an *unroll-and-jam* transformation in order to uncover the vector parallelism available in the external loop. Figure 4 illustrates this transformation for the example in Figure 1. It consists in unrolling the external loop and fusing the resultant instances of the inner loop. The unrolling factor, as in the original SLP compiler algorithm, corresponds to the sws.

To improve SLP exploitation, we have also added to the SLP compiler algorithm both *loop peeling* and *dynamic alignment detection* techniques [9,7,1].

One of the keys to success of the SLP algorithm is its ability to seed the initial *PackSet* with pairs of statements that imply accesses to adjacent memory locations. This translates into a reduction in the number of load instructions and enables the compiler to find vector candidates that are already packed in memory. These adjacent memory accesses can also be found in the basic block generated by the process described in Figure 4. Thus we do not modify the first phase.

However, we modify the original ordering performing the *combination* stage just after the *adjacent memory identification*. In this way, at the end of the *combination* phase the compiler has a

```
for i=0 to N-1
   for j=0 to N-3
      A[i,j+1] = α*A[i,j]+β*A[i,j+2];
           a) Original code

for i=0 to N-1 by 2
   for j=0 to N-3
      A[i,j+1] = α*A[i,j]+β*A[i,j+2];
   for j=0 to N-3
      A[i+1,j+1] = α*A[i+1,j]+β*A[i+1,j+2];
           b) External loop unroling

for i=0 to N-1 by 2
   for j=0 to N-3
      A[i,j+1]   = α*A[i,j]  +β*A[i,j+2];
      A[i+1,j+1] = α*A[i+1,j]+β*A[i+1,j+2];
           c) Jam

for i=0 to N-1 by 2
   for j=0 to N-3 by 2
      A[i,j+1]   = α*A[i,j]    + β*A[i,j+2];
      A[i+1,j+1] = α*A[i+1,j]  + β*A[i+1,j+2];
      A[i,j+2]   = α*A[i,j+1]  + β*A[i,j+3];
      A[i+1,j+2] = α*A[i+1,j+1]+ β*A[i+1,j+3];
           d) Inner loop unrolling
```

Figure 4. *Unroll-and-jam* plus inner loop unrolling for the example described in Figure 1 ($sws = 2$).

PackSet($\mathcal{P}_0$) that only contains *Packs* of statements that access adjacent memory locations. In the code generation phase, each of these *Packs* can be translated to a vector load. For the same reasons as in the original SLP compiler algorithm, it is guarantied that these *Packs* do not exceed the sws and will not cross alignment boundaries.

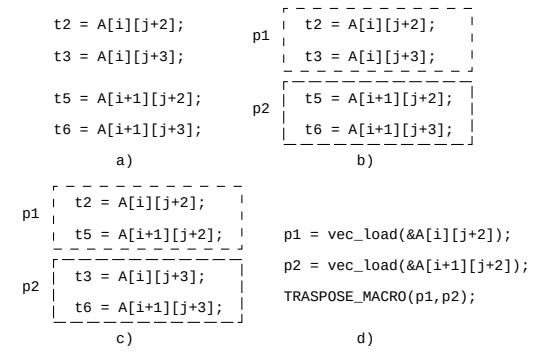

Figure 5. An example of *Pack Transposition* phase and the respective vector code generation for the example in Figure 2 ($sws = 2$).

As a result of the *unroll-and-jam* transformation followed by the inner loop unrolling, we have in the basic block sets with sws equivalent *Packs*. These sets contain statements that perform the same computations on different rows of the arrays (Figure 5b). We will refer to a set of these equivalent *Packs* as a *Group*.

Each *Group* can be seen as a $sws \times sws$ matrix. We introduce a new *Pack Transposition* phase, in order to pack together statements that operate on data elements from different rows. It consists in transposing each of the *Groups* in the *PackSet*. The process is described in Figure 5 for the final

basic block in Figure 4. A new *PackSet* ($\mathcal{P}_1$) is then constructed.

In the code generation phase, this *Pack Transposition* translates into a set of shuffling operations that transform each of the superwords processed by a *Group* in $\mathcal{P}_0$ to the corresponding superwords consumed by the new *Packs* in $\mathcal{P}_1$ (Figure 5d). This process can be done if multidimensional arrays are padded in the lowest dimension, which guaranties constant alignment among rows [8]. As we will show, the shuffling overhead is by far compensated by an increase in the number of short vector instructions finally generated. This new stage is an example of how the packing of elements can be steered according to a dependence analysis.

Finally, the new *PackSet* ($\mathcal{P}_1$) is considered as seed for the *PackSet Extension* and *Scheduling* phases of the original SLP compiler algorithm, with one slight difference, the *PackSet Extension* phase has to work on *Packs* not on *Pairs* of statements.

5. Performance results

As computing platform to evaluate our methodology we have used a 2.4GHz Pentium 4 (768MB 233MHz DDR, 512kB L2) and a 1.42GHz G4 (1GB 167MHz DDR, 512kB L2). As back-ends, we have used the Intel C/C++ compiler (v8.1) and the Altivec Interface of the gcc-3.3. In all cases we have used single precision floating-point as default datatype, which implies sws=4 in both platforms.

In the following subsections we analyzed in detail the performance achieved on our motivating example (the synthetic kernel introduced in Figure 1) and a real kernel extracted from the JPEG2000 (namely, the horizontal filtering of its intra-component transform). In both cases, the automatic vectorization capabilities of the Intel compiler fail to extract the available parallelism.

5.1. Synthetic Kernel

In this example, the original SLP compiler also manages to generate some SIMD code. Its performance is shown in Tables 1 and 2. Given that loop unrolling improve performance by itself, we have analyzed separately this contribution to isolate the benefits of the SLP extraction. As can be noticed, results are qualitative different in both platforms. On the G4, the benefits of the Larsen compiler are negligible and all the performance improvements are achieved by the loop unrolling. In contrast, on the Intel Pentium this loop transformation degrades the performance, although the vectorial loads introduced by the SLP compiler counteract this effect, achieving moderate speedups (between 7% and 12%) over the original code.

Tables 3 and 4 illustrate the benefits achieved by our methodology. As expected, the *unroll-and-jam* transformation improves performance (UJ columns). The other columns show the speedups achieved by SLP and our methodology (denoted as PT-SLP: *Pack Transposition SLP*) over this optimized scalar code. As can be noticed, *unroll-and-jam* does not fit well with the original SLP compiler (SLP+UJ columns), especially on the Intel platform , in which their combination translates into noticeable performance slowdowns.

Our methodology always manages to achieve additional speedups: the overheads introduced in the *Pack Transposition* phase are by far compensated by the additional SLP extracted. Results are qualitative similar on both platforms (i.e. our approach is more robust that the original SLP compiler). For small problem sizes the speedups are close to the ideal values, whereas for the largest problems, in which this benchmark becomes memory bounded, the benefits of the SLP extraction decrease. This behavior highlights the strong influence of the memory hierarchy in the SLP exploitation.

5.2. Intra-component Transform

In this real application, the SLP compiler fails to extract the available SLP, and hence Tables 5 and 6 only show the benefits of the *unroll-and-jam* transformation and the overall speedups achieved by our methodology. Given that *unroll-and-jam* can degrade performance (especially on the Intel Pentium) in this case, the speedups of our methodology refers to original scalar code.

The qualitative results are again similar in both platforms, and the speedup decreases with the problem size. For large problems, this application also becomes memory bounded and the speedups decrease. Quantitatively, gains are outstanding on the G4 platform, in which the speedups for the small problem sizes match the ideal values.

6. Conclusions

In this paper we have introduced a novel compiling methodology, denoted as PT-SLP, that efficiently extracts SLP in some applications in which current compilers fail to generate efficient short vector code.

Our methodology clearly outperforms the original SLP compiler and also exhibits a more robust behavior. Speedups are qualitatively similar on the target architectures, although quantitative results has been much better on the G4 platform. The speedups are higher and close to the ideal values for small problem sizes (around 4 on the G4 and 3 on the Pentium4), and they decrease for large problems since the target algorithms are memory bounded.

These promising results lead us to envision new methodologies for SLP extraction based on improving the original SLP compiler with additional superword recombination stages.

References

[1] A. Bik, M. Girkar, P. Grey, and X. Tian. Efficient exploitation of parallelism on pentium iii and pentium 4 processor-based systems. *Intel Technology Journal*, 2001.

[2] D. Chaver, C. Tenllado, L. Piñuel, M. Prieto, and F. Tirado. 2D wavelet transform enhancement on general-purpose microprocessors: Memory hierarchy and simd parallelism exploitation. In *Int. Conf. of High Performance Computing (HiPC)*, pages 9–21, 2002.

[3] S. Fuller. Motorola's AltiVec technology. Technical Report ALTIVECWP/D, MOTOROLA, 1998.

[4] Intel. Intel architecture optimization. reference manual. http://developer.intel.com.

[5] Intel. Intel architecture software developer's manual. http://developer.intel.com.

[6] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[7] A. Krall and S. Lelait. Compilation techniques for multimedia processors. *Int. Journal on Parallel Programing*, 28(4), 2000.

[8] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proc. of the ACM SIGPLAN 2000 Conf. on Programming language design and implementation (PLDI'00)*, pages 145–156, New York, NY, USA, 2000. ACM Press.

[9] S. Larsen, E. Witchel, and S. Amarasinghe. Techniques for increasing and detecting memory alignment. Technical Report MIT-LCS-TM-621, MIT, USA, 2001.

[10] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley, Massachusetts, USA, 1991.

Synthetic Kernel. Speedups achieved by the original SLP compiler on the Pentium 4 platform. LU and SLP stand for the contribution of loop unrolling and the overall speedup respectively.

size	LU	SLP
128	0.843	1.116
256	0.847	1.116
512	0.879	1.1
1024	0.883	1.075
2048	0.893	1.097
4096	0.902	1.079
144	0.844	1.116
288	0.849	1.12
576	0.86	1.083
1152	0.881	1.071
2304	0.896	1.078
4608	0.902	1.083

Table 3
Synthetic Kernel. Speedups achieved by the SLP and PT-SLP compilers on an Pentium4 platform. Speedups are calculated over an optimized scalar version that already incorporates *unroll-and-jam*.

N	UJ	SLP + UJ	PT-SLP
128	2.154	0.63	3.079
256	2.036	0.655	2.734
512	1.756	0.718	1.826
1024	2.176	0.625	1.491
2048	2.155	0.628	1.474
4096	2.151	0.629	1.474
144	2.171	0.621	3.267
288	2.127	0.624	3.212
576	1.849	0.696	1.77
1152	2.158	0.631	1.492
2304	2.131	0.631	1.516
4608	2.16	0.63	1.501

Table 5
Intra-component transform. Speedups achieved by *unroll-and-jam* and our PT-SLP compiler on the Pentium 4 platform.

N	UJ	PT-SLP
128	1.176	2.277
256	1.175	2.318
512	0.839	1.294
1024	1.117	1.418
2048	1.135	1.435
4096	1.112	1.43
144	1.193	2.267
288	1.121	2.104
576	0.885	1.342
1152	1.126	1.46
2304	1.099	1.456
4608	1.113	1.472

Table 2
Synthetic Kernel. Speedups achieved by the original SLP compiler on the G4 platform. LU and SLP stand for the contribution of loop unrolling and the overall speedup respectively.

size	LU	SLP
128	1.56	1.598
256	1.593	1.66
512	1.385	1.452
1024	1.35	1.418
2048	1.35	1.418
4096	1.352	1.419
144	1.562	1.597
288	1.598	1.665
576	1.365	1.433
1152	1.351	1.419
2304	1.348	1.418
4608	1.351	1.419

Table 4
Synthetic Kernel. Speedups achieved by the SLP and PT-SLP compilers on an the G4 platform. Speedups are calculated over an optimized scalar version that already incorporates *unroll-and-jam*.

N	UJ	SLP + UJ	PT-SLP
128	3.071	1.008	4
256	3.132	1.014	4.078
512	2.333	0.979	1.461
1024	2.116	0.98	1.326
2048	1.932	0.979	1.262
4096	1.772	1.016	1.256
144	3.062	1.013	4.075
288	3.167	1.01	4.152
576	2.254	0.984	1.378
1152	2.09	0.993	1.322
2304	1.93	0.994	1.255
4608	1.716	1.059	1.293

Table 6
Intra-component transform. Speedups achieved by *unroll-and-jam* and our PT-SLP compiler on the G4 platform.

N	UJ	PT-SLP
128	1.079	4.314
256	1.114	4.477
512	1.013	2.34
1024	1	1.985
2048	0.998	1.782
4096	0.99	1.577
144	1.077	4.28
288	1.091	4.136
576	1.006	2.191
1152	0.998	1.951
2304	0.998	1.76
4608	0.99	1.578